SCHOOL OF PHYSICS AND ASTRONOMY
UNIVERSITY OF LEICESTER
Asterix Data Analysis
Programmer Note 001

PROG/001

RJ Vallance and RD Saxton
19-May-1990

Programming Practices

Contents

1 Introduction

These standards relate to programs written or modified for the new X-ray analysis suite. Programs written from scratch should comply with the points in both the 'Immediate' and 'Eventual' categories. Whereas routines which already exist and are imported into ASTERIX need initially only be modified to comply with the 'Immediate' requirements, and can gradually be brought into line with 'Eventual' requirements as time permits in the future.

2 Immediate

2.1 Code layout

All code should be in uppercase. Indentation of either 2 or 3 spaces should be used to show block structuring and spaces and blank lines should be used liberally to improve readability.

2.2 Libraries and naming Conventions for Subroutines

Routines will be put into a small number of libraries. There will be one library for each monolith or package, into which the component applications of the monolith and subroutines specific to these applications will be placed. There will also be a general library, ASTLIB, into which routines that are called by more than one application will be placed. In order to avoid naming conflicts between subroutines belonging to different applications in a monolith the following conventions should be observed:

- Subroutines not specific to an application will reside in the general library, ASTLIB and will be named as follows:

  ```
  prefix_routinename
  ```

  The prefix should be chosen from the standard list given in AST˙LIB:PREFIX.LIS Most routines fall into an obvious category e.g. HDX_ routines for general HDS file handling utilities, BDA_ for binned data access or LIST_ for handling list structures. If a routine does not obviously belong in any category then it should have a prefix of UTIL_. With care these can be kept to a minimum.

- Actual application routines in a monolith or package will be put in the monolith library and called by the application name.

  ```
  e.g. EVBIN, BINSUBSET
  ```

- Where a subroutine or function is specific and subordinate only to one application or routine then its name should be

  ```
  applicationname_subroutine
              or
  routinename_subroutine
  ```

In all of the above cases 'routinename' and 'subroutine' should be alphanumeric only and should be kept to a reasonable length of no more than 12 characters.

2.3 Subroutine/Function arguments

Arguments should be declared and passed in the following order:

```
Input value used but not modified   - Import
Input value used and modified       - Import/Export
Input value not used but modified   - Export
STATUS value
```

All arguments should have type and size declared in the appropriate section of the header. CHARACTER arguments should be declared with length (*) since FORTRAN passes length information anyway. Arrays should have their dimensions passed and declared unless this is potentially zero in which case (*) should be used. In the latter case a logical must also be passed such that access to the non-existent array may be prevented.

2.4 Variable initialisation

All variables should be initialised before use and a value should never be assumed to have been assigned by the system. This is best done at execution time by simple assignment rather than with data statements since if the application is part of a monolith then the initialisation only takes place when that monolith is first loaded, not each time the application is invoked. Values which are to be retained and used by successive invocations of a subroutine should be declared with SAVE for the benefit of other programmers if not for the compiler.

2.5 Error Handling

All subroutines which can fail in any way must have STATUS included in its arguments. In general this will mean any routine which itself calls other routines which return STATUS. It also means any routine which could fail as a result of a previous bad status, for example a routine which accepts mapped data. In this case STATUS must be checked on entry to the routine and control returned immediately to the calling routine in order to prevent 'Access Violation' errors. If STATUS goes bad during the execution of a routine then this should be reported in order to provide a traceback to the offending routine. The following skeleton will in general take care of the above requirements.

```
SUBROUTINE XXXX(STATUS)
          .
          .
IF (STATUS.EQ.SAI__OK) THEN
          .
          .
```

```
          .
    IF (STATUS.NE.SAI__OK) THEN
      CALL ERR_REP('XXXX','from XXXX',STATUS)
    ENDIF

  ENDIF
  END
```

2.6 Version numbering

Every application should be given a version number which should appear in the header. This version should be assigned to a character variable or constant and should have the form:

```
'applic_name Version x.y-z'
```

where 'x.y' is the ASTERIX version number and 'z' is the version of the application within that release. A completely new application should be given the number of the next ASTERIX release, obtained by invoking the application 'RELEASE'. The application should be given a new application version number every time after its initial release that a change is made. The change should entered in the 'History' section of the header.

2.7 History

Every application which changes data in any way must make an entry in the HISTORY structure associated with those data. Adding to the history is simply a matter of making calls to standard routines, but the programmer must decide what gets entered. As a minimum the application name and version number must be entered. Also, where possible, the values of all significant parameters should be included.

2.8 Documentation

- Programmer documentation should be provided in the form of comprehensive in-code commenting. Also where considered necessary, additional details may be inserted into the 'Method' section of the header.
- User documentation should be provided on two levels:
  - Each environment parameter should have a help line entry in the interface file. At the moment this is limited to one line, but this may increase in the future.
  - A text file 'application.HLP' should be provided with each application for inclusion in the ASTHELP system. A standard template for this can be obtained by including

```
HELP_PRO
```

2.9 Using standard data components

Applications should always do something sensible with the minimum components present, which might just be DATA_ARRAY. Applications to which quality is relevant should access it if available and either ignore bad quality values or warn the user of their existence. Errors, where present, should be manipulated under the assumption that they are independent and Gaussian. All access to standard data components (as detailed in PROG_002) should be done via the BDA_ standard access routines and names such as 'AXIS(n)' etc. should not be hard-coded into application code.

2.10 Standard Headers

All applications, subroutines and functions must begin with a standard header or prologue. These are obtained by including the following logical names:

```
APP_PRO      -  for top level routine of application
SUB_PRO      -  for library subroutines or subroutines
                within applications
FUN_PRO      -  same as above but for functions
NODDY_PRO    -  for simple subroutines which don't pass
                STATUS as an argument
```

All applications should begin with a call to the general ASTERIX initialisation routine AST_INIT() and end with AST_CLOSE(). This will ensure the integrity of the internal common blocks used by the various internal sub-systems.

2.11 Propagation rules

(i) If an item is one of the standard objects in the main NDF and the effect of the program on that object is completely known then propagate.

(ii) If an item becomes invalid after an operation, then a warning message should be output and the item should not be propagated.

(iii) If an item is in the MORE extension then it should be propagated by default. If a particular application 'knows' that an extension item has been invalidated then rule (ii) applies.

(iv) If a non-standard item is found in the main NDF it should not be propagated.

3 Eventual

3.1 Application design philosophy

ASTERIX system design philosophy is described more fully in the reference given below but the essential features are listed here. The functionality of individual applications should be simple and essentially 'atomic' in nature, or at least appear so to the user. To achieve this applications with many options should be avoided and where an application needs to follow different paths this should be driven where possible by the input data. User I/O should be minimised; where possible and sensible the application should take reasonable defaults for parameter values unless explicitly overridden

3.2 In-code comments

Code should be liberally commented and all comments should be in lowercase. Standard comment lines should begin with '*', commented-out code should begin with 'C' and commented-out debugging lines with 'D'. End of line comments using '!' are also allowed if neatly aligned.

3.3 TAB

"SET TAB 6" may be used with the editor to cause the TAB to generate 6 spaces at the beginning of a line and this should be used in preference to the raw tab character.

3.4 Line length

Line lengths may not be extended beyond the standard 72 characters for code and comments should not exceed 80. Continuation lines should only be used where necessary eg. to complete the list of subroutine arguments. The only permissible continuation characters are ':' and '&'.

3.5 Line labels

Line labels should be avoided if possible by avoiding the use of FORMAT and GOTO and by using the DO/ENDDO form of loop. Formats should be embedded in the I/O statement as assigned character constants or literals.

3.6 Subroutines and functions

FUNCTION should only be used where all the necessary information for the function to execute is obtained from the calling arguments and where that execution results in an unambiguous return value and no alteration to the argument values. In all other cases SUBROUTINE should be used.

3.7 Variable names

Names in excess of 6 characters are allowed, but may not be longer than 12 characters. Numeric characters should be used sparingly and underscores should be avoided with a maximum of 1 allowable for reasons of genuine readability. No other special characters are allowed. If possible reserved words should be avoided.

3.8 Variable typing

All variables should be explicitly typed, in fact the IMPLICIT NONE statement in the header forces this. Array dimensions should be given in the type statement not in a separate DIMENSION statement. Non-standard types such as BYTE should only be used for compatibility with external data. All type conversion in expressions should be explicit using the REAL, INT and DBLE functions.

3.9 Use of COMMON and STRUCTURE

DEC FORTRAN allows the use of a record type structure similar to that found in C or Pascal. This allows a number of related data items of varying type and size to be grouped under a single name and then passed around different subroutines of an application as a single argument. This makes for a more visible way of passing the information around as compared with the back-door method of pushing it into global variables within COMMON blocks. Some restraint should be employed, however, and in general structures should only be used where the information they contain is to be widely distributed throughout an application. They should not be used just as a shorthand way of passing several arguments between two or three routines - use discrete arguments.

COMMON should only be used where absolutely necessary. There are probably a couple of instances where this might be the case:

- Passing information into subroutines whos' arguments are already defined and cannot be changed eg. a local version of a low-level subroutine within a package.
- Passing information between component applications of a monolith where this cannot be achieved by use of the parameter system.

3.10 The following are BANNED

```
ASSIGN
ENTRY
STOP
PAUSE
RETURN  - except at the beginning of a subroutine in the statement

                IF (STATUS.NE.SAI__OK) RETURN

Logical IF -  except in the above case
Arithmetic IF
Assigned GOTO
DIMENSION
EQUIVALENCE - should be avoided if at all possible.
```

Particularly avoid:

- EQUIVALENCEing a variable in a common block
- Using EQUIVALENCE on items of different data type

3.11 I/O

Terminal I/O should normally be done using the ADAM parameter system PAR_ and message system, MSG_ with the final output being performed by the ASTERIX extension to this , MSG_PRNT. Bulk data are generally stored and accessed via HDS but if files are used then logical unit numbers should be obtained using FIO_GUNIT and freed after use by FIO_PUNIT; these are found in FIO_LIB/LIB. PRINT statements should only be used for debugging.

3.12 Pointers and Mapping

Mapping is a facility provided by VMS which gets around the problem of fitting variable length data into static arrays of fixed length as imposed by FORTRAN. Essentially the area of disk in which the data reside are added to virtual memory, the mapping routines returning the start address in an integer variable. Since FORTRAN has no facilities for dealing with such pointers their use can be hazardous! Mapping and pointers should only be used according to the following rules:

- Mapping should only be done using HDS/ADAM subroutine calls - there should be no explicit reference to VMS mapping service routines.
- The value of a pointer may not be changed.
- The array pointed to may only be manipulated by a lower level subroutine. The pointer should be passed by reference using together with the dimensions of the array. Within the subroutine the array should be dynamically dimensioned unless its length is potentially zero, in which case the (*) format may be used.

3.13 Dynamic Storage

Temporary storage should be obtained using the DYN_ routines in preference to the method of creating a temporary HDS object and mapping.

3.14 Tidying Up

All locators obtained by the parameter system are annulled by ADAM at the termination of an application whether run as a single task or within a monolith. So the cumbersome process of checking the validity of locators and then annulling them can be avoided. However, it is probably neater and safer (from possible future changes in behaviour) to only assume this for locators in the top level routine of an application. All locators created in lower level routines should be properly disposed of when finished with.

All mapped data should be explicitly unmapped when no longer needed.

4 References

ROSAT/GS/UL/85/007	ROSAT coding standards
SGP16	Starlink programming standards
ADAM_DOCS:AED003	Interface files
ADAM_DOCS:AED007	Monoliths
ADAM_DOCS:AED012	Atasks
ADAM_DOCS:AED014	Message and Error systems
ADAM_DOCS:AED015	Parameter system
AST_DOCS:PROG_002	ASTERIX standard data structures
AST_DOCS:PROG_003	BDA_ binned data access routines
AST_DOCS:PROG_004	DYN_ dynamic memory routines